Efficient Dynamic Programming for MapReduce

Coman, Andrei Belulkar, Nikhilesh Uribe, Imanol Macleod, Ian

May 17, 2022

Abstract

Summary/Research project on efficient implementations of dynamic programming recurrences in the MapReduce framework. In particular, we focus on constant/polylogarithmic-round methods, where n denotes the size of the input.

1 Introduction

Parallel computing can be an extremely powerful tool, potentially enabling greater efficiency for a wide variety of well-researched problems in the context of huge datasets that are unable to be realistically processed on one machine. However, parallelization has natural limitations that we introduce below. Firstly, we define the notion of efficient parallel computation as follows:

Definition. *Efficient Parallel Computation: A language L can be decided efficiently in a parallelized model if L can be decided in poly-log time using polynomially many processors.*

Firstly, we consider the space of all problems that are amenable to parallelization by constraining our focus to languages in the complexity class **NC**. We define **NC** in relation to circuit complexity, following the canonical definition [Coo79, AB09]: A language L is in **NC** if there are constants $c, d > 0$ such that L can be decided by a logspace-uniform family of circuits $\{C_n\}$ (called log-space uniform because $\exists$ DTM M that requires logarithmic space to output a description of the circuit C_n on an input 1^n) where C_n has size $O(n^c)$ and depth $O(log^d n)$.

Theorem A computational task is said to have efficient parallel algorithms i.f.f. inputs of size n can be solved using a parallel computer with $n^{O(1)}$ processors and in time $log^{O(1)} n$.

From this theorem it follows that L $\in$ **NC** $\iff$ there exists an efficient parallel algorithm for L. It is worthwhile to note a large open question in complexity theory: $\mathbf{P} \overset{?}{=} \mathbf{NC}$ (it is widely believed that the two complexity classes are not equal, see [AB09]). We now investigate different applications of parallel computing models applied to dynamic programming problems, differentiating between algorithms that require constant number of rounds and algorithms that require polylogarithmic number of rounds of parallel computation. We focus specifically on the MapReduce model of $\mathcal{MPC}$.

2 $O(1)$-round Algorithms

In this section, we will give an intuitive justification for the following two theorems [IMS17]:

Theorem For $\varepsilon, \delta > 0$, $\exists$ a $(1+\varepsilon)$-approximate algorithm solving the Optimal Binary Search Tree problem in $O_{\delta,\varepsilon}(1)$ rounds and $\tilde{O}(n^\delta)$ memory per machine.

Theorem For $\varepsilon, \delta > 0$, $\exists$ a $(1-\varepsilon)$-approximate algorithm solving the Weighted Interval Selection problem in $O_{\delta,\varepsilon}(1)$ rounds and $\tilde{O}(n^\delta)$ memory per machine.

2.1 Optimal Binary Search Tree

The Optimal Binary Search Tree (OBST) problem consists of constructing the BST of minimal cost from a sorted list of keys with associated search frequencies or weights (w). The cost of a search is the number of nodes that respective search traverses, and the cost of the BST is the sum of the costs of all individual searches. The following recurrence is a well-known solution to the OBST problem:

$$\text{OPT}(i,j) = \begin{cases} \min\limits_{i \le k \le j} (\text{OPT}(i,k-1) + \text{OPT}(k+1,j)) + w(i,j), & \forall 1 \le i \le j \le n \\ 0, & \forall i > j \end{cases}$$

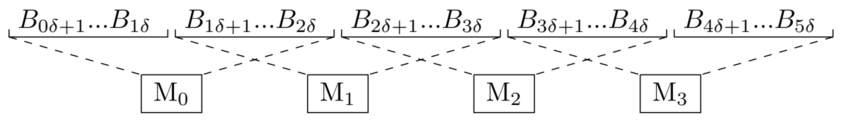
Im et al. [IMS17] have designed a $(1+\varepsilon)$-approximate algorithm that runs in $O(1/\delta)$ rounds, uses $n^{1-\delta}$ machines, $\tilde{O}(n^{\delta})$ space per machine and $\tilde{O}(n)$ space overall. Their solution makes use of two key observations: (1) The cost of optimal solution changes very little if we add or remove small-frequency elements from the input list; thus, one does not need to solve the problem over the exact input, but can instead solve a "coarser" version; this leads to the definition of the Hierarchical Block System below; (2) the costs of the intermediary trees can be discretized, such that the dynamic-programming table OPT from the above recurrence can be compactly represented.

We begin by describing the **Hierarchical Block System**: by a block $B = (s_B, t_B)$, we understand the contiguous selection of the input elements with indices between s_B and t_B. A block system $\mathbb{B} = \{B_1, B_2, ...\}$ is a collection of such blocks which form a partition of the input. An l-block system satisfies the additional requirement that the weight of any two adjacent blocks exceeds l. A Hierarchical Block System is a series of progressively coarser block systems. More formally, let $l_0 = \varepsilon/10n^2$ and $l_h = l_0 n^{\alpha h}$, $\forall h$, where $\alpha := \delta/4$ be the widths of the block systems that the algorithm will iterate through (i.e. the first block has width l_0 and each block system is n^{α} times wider than the previous one). The algorithm first constructs the trivial block system $\mathbb{B}_0 = \{\{0\},\{1\},...,\{n\}\}$. Then, $\forall h$, $\mathbb{B}_h$ (an l_h-block system) is a "refinement" of $\mathbb{B}_{h+1}$. To construct $\mathbb{B}_{h+1}$, we greedily aggregate as many blocks from $\mathbb{B}_h$ as possible without exceeding l_{h+1}. For each block system, each machine solves the problem locally considering only the blocks associated to it and treating that each block is an individual element. Then, the block system is made coarser and the new blocks are redistributed to the machines as individual elements themselves. At every transition from a block system to another, the algorithm incurs multiplicative error by treating the elements of a block as a single element. This error can then be bounded to obtain a $(1+\varepsilon)$-approximation. The parameter α controls the rate at which the blocks grow in size. For α defined as above, it is immediate that, after $O(1/\delta)$ block systems, the entire input gets reduced to a single block.

The second technique is cost-discretization. More specifically, instead of computing the entries $\text{OPT}(i,j)$ for each pair of indices i,j, we define, for every block system $\mathbb{B}_h$, the following quantity: $\text{last}_h(a,c)$ is the largest index $b \ge a$ such that we can construct a tree of cost at most c from blocks $B_a, ..., B_b$. We then only consier c of the form $v_q = \frac{\varepsilon}{10n^2}(1+\varepsilon_0)^q$, where $\varepsilon_0 = \varepsilon/100\log n$. This way, from the original $O(n^2)$-space representation, we arrive at an $O(n \log n)$-space approximation of the original costs. The pairwise cost can then be recovered as follows:

$$\text{cost'}(B_a, B_b) = \begin{cases} 0, & \text{if } a > b \\ \frac{\varepsilon}{10n^2}(1+\varepsilon)^{\underset{\text{last}_h(a,r) \ge b}{\arg\min} r}, & \text{otherwise} \end{cases}$$

An important remark is that a single machine cannot compute the entries of last_h starting from every block B_a associated to it (because, for the rightmost blocks, the necessary information will exceed the bounds of the machine). Hence, we overlap the information sent to the machine as follows:


$B_{0\delta+1}...B_{1\delta}$
$B_{1\delta+1}...B_{2\delta}$
$B_{2\delta+1}...B_{3\delta}$
$B_{3\delta+1}...B_{4\delta}$
$B_{4\delta+1}...B_{5\delta}$
M_0
M_1
M_2
M_3

That is, we send the information of blocks $B_{1...2\delta}$ to the first machine, $B_{\delta+1...3\delta}$ to the second one and so on. This way, we allow each machine to compute the entries of last_h corresponding to the first half of the blocks associated to it, so that the entire table is filled out correctly.

Now, we describe how the entries of last_h are computed for a block system $\mathbb{B}_h$. The following definition measures, for cost' obtained from last_h, the range for which approximation is reliable.

Definition We say that cost'_h is updated up to weight v if:

$$\text{cost'}_h(B_a, B_b) \leq (1+\varepsilon_0)^{10h\alpha\log n}\text{OPT}(s_{B_a}, t_{B_b})$$

for all blocks $B_a, B_B \in \mathbb{B}_h$ such that $w(B_a, B_b) \leq v$.

Immediately after switching to a coarser block system, the width of this reliable approximation is relatively small. However, starting from a small such width, the approximation can then be extended as follows. We first define "cost" according to the following recurrence. This is similar to the original definition of OPT in the sense that it chooses the root of the tree and constructs the problem recursively.

$$\text{cost}(B_a, B_b) = \min_{\substack{a\leq b_0\leq b \\ w(B_a, B_{b_0-1})<3w(B_a,B_b)/4 \\ w(B_{b_0+1}, B_b)<3w(B_a,B_b)/4}} \begin{cases} \text{cost'}(B_a, B_{b_0}) + \text{cost'}(B_{b_0}, B_b) + w(B_a, B_b), & \text{if } |B_{b_0}| > 1 \\ \text{cost'}(B_a, B_{b_0} - 1) + \text{cost'}(B_{b_0} + 1, B_b) + w(B_a, B_b), & \text{if } |B_{b_0}| = 1 \end{cases}$$

Then, we propagate the information back to the compact representation:

$$\text{last}_h(a, q) = \max_{\substack{\text{cost}_h(B_a, B'_b)\leq v_q \\ \forall b'\leq b}} b$$

Lemma If cost' is originally updated up to width $l_h n^{2\alpha}$, after the application of the above steps, it is updated up to width $\frac{4}{3}l_h n^{2\alpha}$.

Proof Let $B_a, B_b \in \mathbb{B}_h$ be two blocks such that $w(B_a, B_b) \geq l_h n^{2\alpha}$, k be the optimal root and $B_u \in \mathbb{B}_h$ be the block containing k. It can be shown that k satisfies the following: $w(s_a, k-1) \leq 2w(s_a, t_b)/3$ and $w(k+1, t_b) \leq 2w(s_a, t_b)/3$ (i.e. the choice of the root is never too close to the end-blocks). Therefore:

$$w(s_a, s_u - 1) \geq \frac{w(s_a, s_b)}{3} - l_h \geq \left(\frac{1}{3} - \frac{1}{n^{2\alpha}}\right) w(B_a, B_b) \geq \frac{1}{4}w(B_a, B_b)$$

and, similarly, $w(t_u+1, t_b) \geq w(B_a, B_b)/4$. Therefore $\frac{w(k,t_u)}{w(s_a,t_u)} \leq \frac{l_h}{w(s_a,t_b)/4} \leq \frac{4}{n^{2\alpha}}$. This implies that, if we consider the blocks as individual elements and take B_u to be the root of the tree, we only neglect some weight that is very small compared to the weight of the entire tree. For this reason (recall the initial observations), the overall cost is only affected slightly. Using this, it can be formally shown that $\text{OPT}(s_a, t_u) \leq (1+\varepsilon_0)OPT(s_a, t_b)$, which leads to the conclusion that, by applying the above steps and approximating the blocks as individual elements, we can "extend" the range of reliable approximation while incurring $(1+\varepsilon_0)$ multiplicative error. Since cost' is itself an approximation of last_h, the overall error for the step is $(1+\varepsilon_0)^2$. The 4/3 constant comes from a technical constraint that we overlook in this overview. For a detailed explanation, see Lemma 3.10 in the original paper [IMS17]. □

If we apply the same step $\lceil \log_{4/3} n\rceil \leq 3\alpha\log n^\alpha$ times, we will incur error $(1+\varepsilon_0)^{6\alpha\log n} \leq (1+\varepsilon_0)^{10\alpha\log n}$ and cost' will be updated up to weight $l_h n^{3\alpha} = l_{h+1}n^{2\alpha}$. So, after the application of these steps, the approximation is reliable for a range large enough to serve as the base step of the next block system in the hierarchy. The transition to the coarser block system $\mathbb{B}_{h+1}$ proceeds as follows. For every $B_x = (s_x, t_x) \in \mathbb{B}_{h+1}$, let $B_a \in \mathbb{B}_h$ be the first block considered in the aggregation of B_x. For any q such that $l_h n^{2\alpha} \leq v_q \leq l_h n^{3\alpha}$, let $B_y \in \mathbb{B}_{h+1}$ denote the block containing $B_{\text{last}_h(a,q)} \in \mathbb{B}_h$. Then:

$$\text{last}_{h+1}(x, q) = \begin{cases} y, & \text{if } t_y = t_{\text{last}_h(a,q)} \\ y - 1, & \text{if } t_y \neq t_{\text{last}_h(a,q)} \end{cases}$$

In other words, this step just formalizes the approximation of the new blocks as individual elements. Considering this definition, it is true that:

$$\text{cost'}_{h+1}(B_x, B_y) = \text{cost'}_h(B_a, B_{\text{last}_h(a,q)}) \leq (1+\varepsilon_0)^{10h\alpha\log n}(1+\varepsilon_0)^{10\alpha\log n}OPT(s_{B_x}, t_{B_y})$$

So, by the definition above, cost'_{h+1} is updated up to weight $l_{h+1}n^{2\alpha}$. This allows us to repeat the expansion steps until we are ready to change the block system again. This completes the intuitive justification of the algorithm. In short, the algorithm is just a repeated application of these "expansion" steps, interleaved with occasional shifts to coarser blocks.

2.2 Weighted Interval Selection

The Weighted Interval Selection problem (WIS) considers a collection of intervals $\{I_i = (a_i, b_i)\}_{i \in [n]}$ with associated weights $\{w_i\}_{i \in [n]}$ and consists of choosing a subset of disjoint intervals with maximal total weight. As in the case of OBST, there exists a well-known dynamical-programming approach:

$$A(i) = \max\{A(i+1), w_i + A(j)\}, \text{ where } j := \underset{b_i < a'_j}{\arg\min}\, j', \forall i \leq n$$

The solution proposed by Im et al. [IMS17] is similar in spirit to the one for the OBST, in the sense that it relies on solving progressively coarser instances of the original problem. While, for OBST, the blocks were considered as individual nodes, in this case, a group of intervals will be considered as a singular interval. More formally, we first order the intervals in increasing order of their starting time and distribute them equally among the machines. We call a subset of disjoint intervals D a block, whose starting time is the start of the leftmost interval, and whose end time is the end of the rightmost interval. A block *spans* machines $k, k+1, ..., k'$ if the earliest interval in D starts on machine k and ends on machine k'. We call an interval *crossing* if it starts and ends in different blocks. An l-block is a block containing at most l crossing intervals.

Lemma For any $L \geq 2$, there exists a $(1 - 2/L)$-approximate solution consisting L-blocks spanning disjoint machines.

Proof Consider an optimal solution and greedily aggregate groups of L consecutive crossing intervals. Every group except the last one will contain exactly L such intervals. Now, remove the lightest interval from each group except for the last one. By this choice, the weight of the solution decreases by at most a factor of $1/L$. Now, the resulting solution consists of $2L$-blocks that span disjoint machines. By scaling L, the proof is complete. Note that if we then let $L = 2/\varepsilon$, this becomes a $(1-\varepsilon)$-approximation. $\square$

Now, the algorithm proceeds in a similar manner to the OBST one described above, in the sense that it computes an efficient representation of the solution for l-blocks, and then considers these l-blocks as individual intervals which constitute a new instance of the original problem. Because all the crossing intervals in the solution are considered within these blocks, the only information relevant for an l-block is the machine on which it starts and its weight. Hence, the new instance of the problem will have $O(n^{1-\delta} \log n)$ intervals (following a discretization of the weights). Thus, we only require $O(1/\delta)$ such reductions.

Because of the significant overlap with the previous algorithm, we will limit our description to an overview of how the solution for l-blocks is created accross machines. The algorithm constructs incrementally larger $(2^i - 1)$-blocks. To transition from $(2^i - 1)$-blocks to $(2^{i+1} - 1)$-blocks, we must first note that any $(2^{i+1} - 1)$-block can be decomposed into a $(2^i - 1)$-block, followed by a crossing interval, followed by another $(2^i - 1)$-block. So, we discretize the weights logarithmically (as in the OBST case), and then, for every starting interval I and for every triplet of weights μ_1, μ_2, μ_3, we consider the shortest $(2^i - 1)$-block of weight μ_1 starting at I, followed by the earliest-ending crossing interval J of weight μ_2, followed by the earliest-ending $(2^i - 1)$-block of weight μ_3 starting after J ends. The combination of these blocks and crossing intervals will constitute a candidate for the best $(\mu_1 + \mu_2 + \mu_3)$-weight $(2^{i+1} - 1)$-block starting at I.

2.3 Open Questions

- Donald E. Knuth [Knu71] and Frances Yao [Yao80] showed that the naive OBST recurrence is a particular case of what is termed the Quadrangle Inequality and, therefore, it admits an $O(n^2)$-time solution. A natural question is whether the parallelism of the algorithm proposed by Im et al. is also a consequence the same Quadrangle Inequality, and if the same results from the sequential model have an analogue in Map Reduce.
- Is there an optimal solution to the 2D Weighted Interval Selection problem (i.e. each interval occupies a space in two dimensions; no two intervals can intersect in any of the dimensions). Solving this problem could yield efficient $(1+\varepsilon)$ approximations for "difficult" problems, such as the 2-way Partition (to the best of our knowledge, no such approximation is yet known for this problem in particular).

3 $O(\log n)$-round Algorithms

3.1 Approximate Edit Distance using Quantum, MapReduce and Dynamic Programming [BEG+18]

Statement of the paper's findings

The paper gives a $(3+\epsilon)$ approximation for edit distance using a MapReduce algorithm with sub-linear number of machines, sub-linear memory per machine and a logarithmic number of rounds.

Descriptive Overview of the Algorithm

The algorithm approximates the edit distance by solving several instances of the δ-bounded edit distance in parallel and then combining the solutions to create an overall approximation.

Definition (δ-bounded edit distance)
Input: two strings s_1, s_2 and a real number $0 \leq \delta \leq 1$.
Guarantee: $edit(s_1, s_2) \leq \delta(|s_1| + |s_2|)$
Output (with an approx. factor $\alpha > 1$): a sequence of operations with cost at most $(\delta \cdot \alpha)(|s_1|+|s_2|)$ that transforms s_1 and s_2

The above approximation problem can be efficiently parallelized with different values of δ that take upon a specific proportion of the input length. Specifically the researchers set $\delta = n^{-\alpha}$, so that there will a logarithmic order of δ-bounded edit distance sub-problems that need to be computed.

There exist two strategies for solving the sub-problems of varying sizes of δ. As mentioned before, there is essentially a logarithmic (in n) amount of these δ values which the researchers need to compute. These δ-bounded edit-distance sub-problems are computed in parallel. For all such δ-bounded edit-distance sub-problems, a constant number or logarithmic amount of rounds is needed. Hence the overall running time of the algorithm is also logarithmic in order.

The classification of δ is determined by the specific number of machines that are used. When $n^{8/9}$ machines are used, the critical value $\delta^* = n^{-8/27}$.

These two sub-strategies for computing are as follows:

- For large δ ($\geq \delta^*$) values, we harness the strategy of using divide and conquer + DP. This is effective because this method of building windows to estimate the edit distance can be effectively parallelized across several machines. It requires a constant number of rounds of MapReduce computation. This approach is ineffective for small values of δ because there would be too many windows created to efficiently compute an approximation;
- For small δ ($< \delta^*$) values, another algorithm technique known as (min, +) matrix multiplication can be used which runs relatively quickly for small delta because it involves building matrices of dimension as some factor of δ.

Once the researchers have these δ-bounded solutions they can aggregate them to find the optimal approximation.

Subroutines within the algorithm

There are two main subroutines we need to understand for this algorithm:

1. Divide and conquer + DP algorithm solution for δ-bounded distance;
2. (min, +) matrix multiplication.

Divide-and-conquer DP algorithm for δ-bounded distance with large $\delta \geq \delta^*$

The overall idea for this DP technique for approximating δ-bounded edit-distance is to construct "window compatible" transformations. We must first define the notion of a window for a string:

Definition (Window): Given a string instance s_1, the authors define a window size $l = n^{1-\beta}$ of some sub-linear portion of the string s_1. Further we define the notion of a layer $\gamma > 1, \gamma \in N$ which is how many sets of windows we might have for a particular string. The gap size is defined by the window size divided by the number of layers or simply $\frac{l}{\gamma}$.

A visualization of the terms introduced in this definition is shown below:

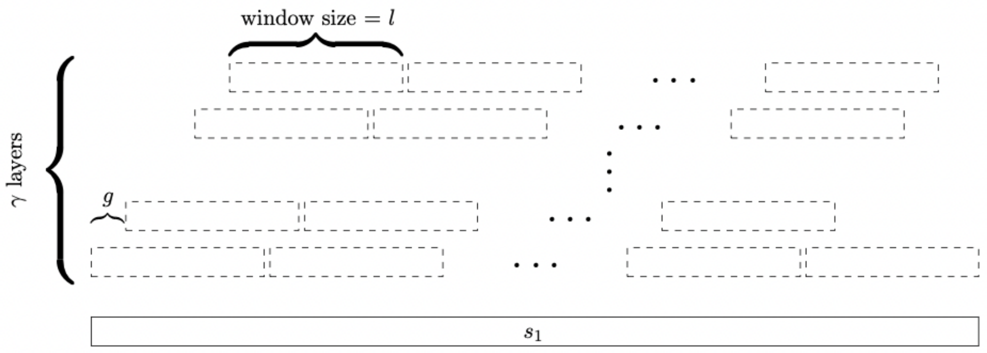


Figure 1: Windows for s_1, with gap size g, window size l and γ layers

The researchers construct such windows based on some intelligent selection of the window size, the number of layers, and the gap for both s_1 and s_2 which we will analyze later. This provides us two sets of windows W_1 and W_2 for strings s_1, s_2 respectively.

Definition (Window Compatible Transformation): A window compatible transformation for some size k is such that there is a pair of sequences of windows from W_1 (computed from s_1) and W_2 (computed from s_2). Call the sequence from W_1 as $\phi =< w_1, ..., w_k >$, and call the sequence from W_2 as $\psi =< w'_1, ..., w'_k >$ These sequences must satisfy three key conditions:

1. No windows in ϕ overlap $\wedge$ no windows in ψ overlap;
2. All original characters is s_2 is also in the windows of ψ;
3. Every original character of s_2 which is in a window w'_i was placed in window w_i prior to the transformation.

The researchers use the conception of "window compatible transformation" to run the naive edit distance DP on smaller sub-strings of the problem. This obviously incurs some accuracy loss, and the notion of window compatible transformations helps bound this loss. Furthermore, the windows provide a natural structure for the parallelization of the algorithm.

Lemma 1. *∃ a window compatible transformation of s_1 into s_2 with respect to W_1 and W_2, that for each matching of windows $w_i \in W_1$ that matches to w'_j in W_2 their indices do not differ by more than $\lceil \delta n/g \rceil$, where g is the gap size.*

Proof sketch. *For the δ-bounded problem $edit(s_1, s_2) \leq \delta n$. Hence, for any v_i and v'_i positions for the first characters of matching windows w_i, w'_j cannot differ by more than δn. Since the gap value is g this must mean the number of matching windows cannot differ by a value of $\lceil \delta n/g \rceil$.*

For a particular value of δ we compute two sets of windows W_1 and W_2 for both strings. These windows can be computed using the values of γ, β, l and can be done in an intelligent way which we discuss below.

Lemma 2. *The number of operations is at most $(3\delta + 1/\gamma)n + 2l$ for this specific window.*

The above result can be proved through construction. The researchers used a sequence M of edges in the optimal cost solution to construct an upper bound for the cost of a window compatible transformation through taking the optimal solution and then determining how many operations the window-compatible transformation sub-problem needs to be solved. The rest of the proof follows relatively straightforwardly through considering gap-size between the windows and the largest possible deviation from the optimal solution to the δ-bounded edit distance problem.

Provided this result for the approximation, we can then see how we select the parameterization for the windows to guarantee a reasonable running time. The parameters of β and γ are used to construct windows of length $l = \lfloor n^{1-\beta} \rfloor$ with a gap size $g = \lceil l/\gamma \rceil$. Since the researchers require a $(3 + \epsilon)$ approximation, they set $\gamma = \frac{1}{\delta\epsilon}, \beta > \alpha$. Accordingly, we substitute these values into the maximal number of operations $(3\delta + 1/\gamma)n + 2l < (3\delta + \delta\epsilon)n + 2n^{1-a} = \delta(3 + \epsilon)n + 2n^{1-a}$, which provides a $(3 + \epsilon)$ approximation to the δ-bounded edit distance problem.

Algorithm 1 $\mathcal{MPC}$ Map Reduce Subroutine for a specific value of $\delta \geq \delta^*$

```
Require: $\delta, \alpha, s_1, s_2, x$
  Set $\gamma = \frac{1}{\delta\epsilon}, \beta > \alpha, l = \lfloor n^{1-\beta} \rfloor, g = \lceil \frac{l}{\gamma} \rceil$

  Create windows $W_1$ for $s_1$ and $W_2$ for $s_2$, with length $l$, and gap $g$ as defined above

  Allocate $(1/\epsilon^2)n^{\alpha+2\beta-x}$ pairs to each one of the $n^x$ machines. These pairs are all the pairs that are
  within a "useful gap" where the windows are not further than $\lceil \delta n/g \rceil$

  In the first round of mapReduce, on each machine m use the naïve DP algorithm to compute distances
  between all pairs of windows allocated to the machine

  In the second round of mapReduce, use only one machine to combine the edit-distance results from
  all the local computations to obtain a $\delta$-bounded edit distance value.
```

Runtime analysis: In the first round of MapReduce the researchers prove that number of windows in each string is $\leq n\gamma/l = O((1/\epsilon)n^{\alpha+\beta})$. This is simple to see in that the number of characters each strings can have at most is n. For one layer of windows given a particular window length the number of windows is $\frac{n}{l}$, given that we have γ layers we multiply this value by γ to get the total maximum number of windows for the string which is $\frac{\gamma n}{l}$

Further from **lemma 1** we know that the number of useful windows is limited by the fact that for window compatible transformations, matching windows are a maximum of $\lceil \delta n/g \rceil$ indices apart.

This means for each window we would only have to analyze $2\lceil \delta n/g \rceil + 1$ windows to ensure that we check all useful pairs of windows, computed by analyzing the distance from either side.

Hence the total number of windows we would need to check would be provided by

$$\min(|W_1|, |W_2|) \cdot (2\lceil \delta n/g \rceil + 1) = O((1/\epsilon^2)n^{\alpha+2\beta})$$

Distributing these windows among some $O(n^x)$ machines leads to every machine getting $O((1/\epsilon^2)n^{\alpha+2\beta-x})$ pairs per machine.

Each machine runs the naïve-DP edit distance solution. We know the length is $l = \lfloor n^{1-\beta} \rfloor$ for each window hence the run time for each pair of windows would be $O(l^2) = O(n^{2-2\beta})$ and requiring memory for each pair of windows $O(l) = O(n^{1-\beta})$.

Hence memory per machine:

$$O(\text{windows/machine} * \text{memory/window}) = O((1/\epsilon^2)n^{\alpha+2\beta-x} \cdot n^{1-\beta}) = O((1/\epsilon^2)n^{1+\alpha+\beta-x})$$

Hence processing time per machine:

$$O(\text{windows/machine} * \text{time/window}) = O((1/\epsilon^2)n^{\alpha+2\beta-x} \cdot n^{2-2\beta}) = O((1/\epsilon^2)n^{2+\alpha-x})$$

In the second round of map reduce the researchers inspect every single edit-distance computed for all pairs of windows and compile the best solution.

The amount of memory that we need is simply $O((1/\epsilon^2)n^{\alpha+2\beta})$, further since we have to compute the edit distance between all pairs this will incur the same complexity $O((1/\epsilon^2)n^{\alpha+2\beta})$

(min, +) matrix product for solving δ-bounded distance with small $\delta \leq \delta^*$

Definition (min, +) matrix multiplication, also known as the distance product, is an operation on matrices. Given two $n \times n$ matrices $A = (a_{ij})$ and $B = (b_{ij})$ their distance product $C = (c_{ij}) = A \star B$ is defined as such: $c_{ij} = \min_{k=1}^{n} a_{ik} + b_{kj}$

The researchers in this part of the algorithm use a result from another paper that provides a fast matrix multiplication result in the MPC paradigm for (min, +) matrix multiplication.

Corollary 1. *(min, +) multiplication of n^a matrices of size $n^b \times n^b$ can be computed in $2\lceil a\log_2(n) \rceil$ rounds of MapReduce with n^y machines for any $0 \geq y \geq a + 3b/2$ with memory of $O(n^{2(a+3b-y)/3})$ for each machine. The running time for each machine is $\tilde{O}(n^{a+3b-y})$.*

In order to prove this corollary, we will have to consider the following theorem:

Theorem 1. *For any two $n \times n$ matrices A and B with $0 \leq x \leq 2$, $A * B$ can be computed with $n^{3(1-x/2)}$ machines with $O(n^x)$ memory in $1 + \lceil (1 - x/2)/x \rceil$ mapReduce rounds [HLSS19].*

Proof sketch. *Consider dividing the the solution matrix C, $C = A \star B$ into sub-matrices of size $n^{x/2} \times n^{x/2}$; this means we have $n^{2(1-x/2)}$ sub-matrices of size $n^{x/2} \times n^{x/2}$. Further, we can compute each c_{ij} with the formula $c_{ij} = \min_{k=1}^{n} a_{ik} + b_{kj}$. Therefore, we assign $n^{1-x/2}$ sub-matrices to each machine. In the first round, each machine outputs these values according to their ranges, and in the next $\lceil (1 - x/2)/2 \rceil$ we compute the $c_{i,j}$ based on the generated values of round 1.*

Corollary 1 is useful for the δ-bounded edit distance problem because many of the sub-problems to the problem for small δ values the researchers proof the following lemma where $d(i, j+1, i', j'+1)$ is the edit distance between two sub-strings $s_1[i, ..., j]$ and $s_2[i', ..., j']$

1) $d(i, j+1, i', j'+1) = min_{i'-1 \leq k \leq j'} d(i, k+1, i', k'+1) + d(k+1, j+1, k'+1, j+1)$

This is true provided that the edit distance of any two strings is the minimum of the edit distances between two corresponding segments of the same strings. We notice that since the optimality statement is of a specific form that will enable (min, +) matrix multiplication of **Corollary 6.2** to come in handy, which we shall now examine mathematically.

The reasoning behind the result in **Corollary 1** is that (min, +) matrix multiplication can be parallelized quite easily through computing smaller instances of the problem on computers and then combining them together. The logic is that the above in equation **1)** can have each individual k computed on a different computer in which the naive DP problem is computed and then aggregated.

The above result enables us to reduce the problem of finding edit distance to a (min, +) matrix multiplication operation. Notice that $edit(s_1, s_2) = d(1, |s_1| + 1, 1, |s_2| + 1)$.

The first **Corollary** of [Ukk85] states the following claim: for computing $d(1, |s_1| + 1, 1, |s_2| + 1)$, is only useful when $|i - i'| \leq edit(s_1, s_2) \wedge |j - j'| \leq edit(s_1, s_2)$. Hence we know since we are analyzing the δ-bounded version of this problem that a similar bound applies: $|i - i'| \leq \delta \cdot edit(s_1, s_2) \wedge |j - j'| \leq \delta \cdot edit(s_1, s_2)$. Therefore we can bound the size of the matrix $D^{1,|s_1|}$ from these inequalities as only having $(2\delta n + 1) \times (2\delta n + 1)$ useful values. The value $d(1, |s_1| + 1, 1, |s_2| + 1)$ is an element of $D^{1,|s_1|}$.

The researchers find the matrix $D^{1,|s_1|}$ by partitioning s_1 into sub-strings and sub-strings for s_2. The naive algorithm is then used to compute the edit distance for these sub-strings creating the sub-matrices that can be used to compute $D^{1,|s_1|}$. These sub-matrices are related to $D^{1,|s_1|}$ through the (min, +) matrix operation. These sub-matrices are of dimensions $(2\delta n + 1) \times (2\delta n + 1)$. Computing all such matrices and combining them together requires a $O(log(n))$ amount of rounds, as follows from **Corollary 1**.

Combining the results from the two previous steps

Hence, the researchers used these two approaches to calculate δ-bounded edit distance for various δ values ranging from $\delta = 0$ to $\delta = (1+\epsilon/3)^k/n$ for $0 \leq k \leq O((1/\epsilon)log(n))$. Further all these δ-bounded edit distance problems are computed in parallel with a scaled amount of machines depending on the difficulty of the problem (we do not focus on the scaling of the machines for the sake of brevity). For $\delta \geq \delta^*$ we run the divide and conquer DP algorithm that requires $O(1)$ rounds in parallel. For the $\delta \leq \delta^*$ we run the (Min, +) matrix multiplication problem which requires $O(log(n))$ rounds. Plugging in the appropriate values for constants in these algorithms yields a running time of $O(n^{2-8/27})$ per machine with a memory of $O((1/\epsilon^2)n^{8/9+\epsilon'})$ in $O(log(n))$ amount of rounds with a factor $3 + \epsilon$ approximation.

3.2 MapReduce Algorithms For Massive Trees [BBD+18]

In this section, we will give an intuitive justification for the following theorem:

Theorem For any given m, there exists an algorithm to solve any (poly log)-expressible problem in $O(\log n)$ rounds of $\mathcal{MPC}$ using m machines, such that with high probability each machine uses a space of size at most $\tilde{O}(n/m)$ and runs an algorithm that is linear in its input size.

We first establish the definitions of binary adaptibility and (poly log)-expressibility.

Definition. *Binary Adaptability: any dynamic program $\mathcal{D}$ that can compute data in terms of a chain of binary operations is binary adaptable, ex:* $\max\{\mathcal{D}(u_1), \mathcal{D}(u_2), \mathcal{D}(u_3)\} = \max\{\mathcal{D}(u_1), \max\{\mathcal{D}(u_2), \mathcal{D}(u_3)\}\}$.

Because our problems are binary adaptable, we can use $\mathcal{T}$ to denote the tree associated with the original problem and $\mathcal{T}^b$ to denote the adapted binary tree associated with the problem.

Definition. *f-expressibility: a binary-adaptable dynamic programming problem $\mathcal{D}$ is $f-$expressive for a function f if the following conditions are met:*

1. The data required to compute a value at any vertex v of $\mathcal{T}^b$ has size $\tilde{O}(f(n_v))$ where n_v is the number of descendants of v.
2. There exists both a compressor $\mathcal{C}$ and a merger $\mathcal{M}$ algorithm. Algorithm $\mathcal{C}$ takes a subtree T of $\mathcal{T}^b$ with $O(1)$ leaves of T for which data is unknown and returns a sketch of size at most $\tilde{O}(f(n_T))$ in time *poly*$(f(n_T))$ such that, when the merger $\mathcal{M}$ receives both the sketch and the dynamic data for unknown leaves, $\mathcal{M}$ returns $\mathcal{D}_P(v)$ in time *poly*$(f(n_T))$.
3. For two disjoint subtrees T_1, T_2 connected by one edge, and for subtree $T = T_1 + T_2$, if there are $O(1)$ leaves of T that are unknown then $\mathcal{M}(\mathcal{C}(T_1), \mathcal{C}(T_2))$ returns $\mathcal{C}_T$ in time *poly*$(f(n_T))$.

We can now introduce a proof for the given theorem above.

Proof sketch: Given a (poly log)-expressible problem P, we first convert the given n-ary tree $\mathcal{T}$ into a binary extension of it $\mathcal{T}^b$ in $O(1)$ rounds and then decompose the binary extension into components to distribute the contents of $\mathcal{T}^b$ across machines in $\tilde{O}(\log n)$ rounds. After this decomposition, every component has at most 2 children, so there are at most 2 leaves in any component that are unknown. Because the number of leaves is constant, we apply the compressor algorithm $\mathcal{C}$ and store data of size at most poly $\log(n)$ for each component. Then in the next step, since there are at most $\tilde{O}(m)$ components, we can send the partial data of all components to one machine that runs the algorithm for merging data, $\mathcal{M}$. (The existence of both $\mathcal{C}$ and $\mathcal{M}$ algorithms stems from P being (poly log)-expressible). This merging of partial data takes polylogarithmic time and space to calculate the answer of the dynamic program for each of the components.

For the purposes of this proof, we are primarily interested in functions f that are poly-logarithmic (many natural graph problems are (poly log)-expressible). We will now give a more rigorous explanation of the process by which a dynamic programming problem can be converted into a binary tree and then decomposed in the $\mathcal{MPC}$ model. Assuming that the problem P is (poly log)-expressible and has been formulated as recursion on the n nodes of the tree $\mathcal{T}$, we give an explicit distributed algorithm for partitioning the tree into $O(m)$ components of size at most $\tilde{O}(n/m)$ across machines. Firstly, we convert the n-ary tree to a binary tree $\mathcal{T}^b$ using the mapping $f : V(\mathcal{T}) \to V(\mathcal{T}^b)$ such that $\forall v \in V(\mathcal{T})$, the ancestors of v are preserved in $\mathcal{T}^b$ and the index of v is equal to the index of $f(v)$ (See Figure 1).

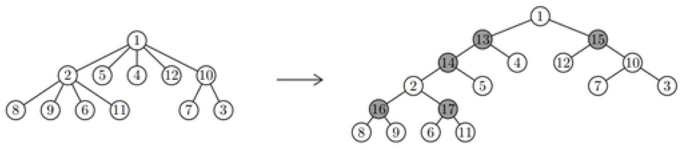


Figure 2: An example of converting a tree to a binary extension of it. The gray nodes denote the added auxiliary vertices and the numbers within the nodes denote their indices.

The conversion from $\mathcal{T}$ to the binary extension $\mathcal{T}^b$ via randomized algorithm requires $O(1)$ rounds of $\mathcal{MPC}$ using m machines and exhibits probabilistic guarantees that each of the m machines uses memory of size at most $\tilde{O}(n/m)$ and runs in time linear to memory size. After this binary extension $\mathcal{T}^b$ has been created, we can decompose it into a set of $\tilde{O}(m)$ components distributed across m machines, where the size of each component is at most $\tilde{O}(n/m)$. The decomposition algorithm ultimately yields a set of components that can be compressed via algorithm $\mathcal{C}$ and merged together via algorithm $\mathcal{M}$ to recover the optimal answer to the original dynamic programming problem.

Definition. *A decomposition on a binary tree is valid if it is a set of $O(m)$ components, where each component contains a subset of the vertices of $\mathcal{T}^b$ and every vertex is contained in exactly one component. Additionally, components are connected by exactly one edge.*

We now give an overview of the decomposition algorithm. The algorithm begins with n components, where each node v of $\mathcal{T}^b$ is a component. It then merges components iteratively until $O(m)$ components remain by selecting a set of components, S, and then merging unselected components to the closest ancestor component that was selected. The criteria for selection at every round ensure that every unselected component has at least one selected ancestor to be merged to, that no component has more than two children, and that the path between a component and its closest selected ancestor doesn't contain any components. Lastly, the algorithm employs probabilistic randomization to ensure that the size of all components is at most $\tilde{O}(n/m)$. This decomposition requires $\tilde{O}(\log n)$ rounds of $\mathcal{MPC}$. We now give the pseudocode for the algorithm:

Algorithm 2 $\mathcal{MPC}$ Binary Tree Decomposition

```
Require: $T^b$
  $C \leftarrow \{v : v \in T^b\}$, $F \leftarrow \emptyset$    ▷ components $C$ are each node of $T^b$, $F$ stores finished components
  while $|C| \geq k \cdot m$ do    ▷ $k$ is a constant s.t. $F$ is $O(m)$
    $S \leftarrow \emptyset$    ▷ represents selected components
    for $c \in C - F$ do
      if $c$ is root or $c$ has two child components or $c$'s parent $c_p \in F$ then
        add $c$ to $S$
      else if $c$ has exactly one child component then
        add $c$ to $S$ with probability .5
    for $c \in C - F - S$ do
      merge $c$ with $c$'s closest ancestor in $S$
    for $c \in C - F$ do
      if $c$ contains at least $n/m$ vertices of $\mathcal{T}^b$ then
        add $c$ to $F$
```

Here is a visual representation of how the algorithm decomposes a binary tree into components.

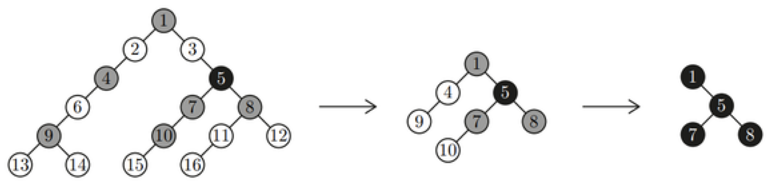


Figure 3: An example running the decomposition algorithm. Each node denotes a component. A gray node denotes a selected component and a black node denotes a completed component.

It should now become clear that, if we can decompose $\mathcal{T}^b$, we will be able to apply the compressor algorithm $\mathcal{C}$ on every component and then use the merger $\mathcal{M}$ on the compressed data sketches to recover the final answer to the dynamic programming problem. The existence of compressor and merger algorithms that enable a dynamic programming problem $\mathcal{D}$ to be f-expressible are non-obvious a priori; thus, we now give examples of specific extensions of this framework to dynamic programming problems that are already likely familiar to the reader.

MPC Warmup: Closest Points

In order to get an intuitive understanding of the power that the $\mathcal{MPC}$ model has in certain classes of problems, let us consider the problem of closest points. Formally the closest points problem is defined as given a set V of n vectors in $\mathbb{R}^d$, and a distance function $f : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$, find the 2 vectors i, j in V which minimize $f(i, j)$. The solutions to this problem in the $\mathcal{MPC}$ universe does not depend on the function f, i.e. you can define any arbitrary distance function and the algorithm will still work. Bateni choose to consider the model where every machine in the $\mathcal{MPC}$ system has access to an oracle which given 2 vectors $x, y \in \mathbb{R}^d$ can compute $f(x, y)$ in $O(d)$ time. Imagine that we wanted to solve this problem in sub linear space, this would mean we could not store all vectors as that would require linear space. So this is what the $\mathcal{MPC}$ model excels at, splitting massive inputs across a variety of machines with sub linear memory. Let m be the number of machines in our $\mathcal{MPC}$ model, and each of these machines can have at s memory. We can define the overall memory $S = s \cdot m$. Let us divide our original input into $k = n^{1/3}$ groups $V_1, \ldots, V_k$, so each group V_i has with high probability $\frac{n}{n^{1/3}} = \tilde{O}(n^{2/3})$ vectors. Next $\forall V_i, V_j$ send them to machine (i, j). Thus we need at least $\binom{k}{2}$ machine which we can upper-bound by saying our number of machines $m = n^{2/3}$. Thus the space that each machine needs to have is at most $O(n/k) = \tilde{O}(n^{2/3})$. Each machine then sends the pair of vectors with the smallest distance to one special machine. Notice that this special machine also only needs $\tilde{O}(n^{2/3})$ space, as it will receive $2 \cdot k$ vectors,which is still within our space bound s (this is crucial as eventually you need that all results end up in one machine, so the way you consistently reduce the dimension of your data in every merging step is critical for any $\mathcal{MPC}$ algorithm). Finally this special machine choose the smallest number out of the k vector pairs. If we want to calculate the total space used we see that $S = sm = (n^{2/3})^2 = \tilde{O}(n^{4/3})$.

This closest points example is a great way to get accustomed to the space conscious splitting and merging process that $\mathcal{MPC}$ consistently uses.

Maximum Weighted Matching

Problem

Given as input a edge-weighted tree T, let w_e denote the weight of edge e, let $n(v)$ be the neighbors of vertex v. The goal is to find a subgraph M of T, such that $\forall v \in M, |(n(v)| = 0$ (i.e no 2 edges can be chosen which share a vertex), and which maximizes $\sum_{e \in M} w_e$.

Since we are given a tree we are guaranteed that there will be no cycles. Next, using what was discussed above we will take this graph T and construct T^b such that we create a binary tree. Using the discussion from above we can take as given that one can construct a T^b, which is a binary tree of T such that every vertex in T is also in T^b. Thus we will know be solving for a subgraph M^b of T^b. Changing the input to create a binary tree allows for much more parrelilzation as you can be confident that any vertex has at most degree 3. However in this new T^b there will be auxiliary nodes (as defined above) in order to preserve the ancestry of the tree (i.e. a nodes closest ancestor which is not an auxiliary node in T^b will also be the nodes' ancestor in T). If there is a directed edge $e = (v, u) \in T^b$ and u is an auxiliary node then $w_e = 0$, else $w_e = w_{e'=(u,v) \in T}$. So since we have defined out new altered binary tree T^b as above we can say that the Maximum Weighted Matching of T^b is equal to the Maximum Matching of T.

First before proof assume that this problem is (poly log)-expressible. Using the theorems from above we automatically can convert the input tree into a binary extension, and then use the decomposition method stated above in order to split the tree into components, which in high probability are evenly sized and thus can fit on machines with $\tilde{O}(n/m)$ space, using $\tilde{O}(\log n)$ rounds. Making the tree binary is crucial as now we are guaranteed that each component can have at most 2 children (and thus we can bound the information that every component needs to store about its neighbors). So after decomposing on m machines, there exists a compressor algorithm which for a partial subtree of the graph will return a polylog(n) sized data for each component, because we assume that our DP problem is (poly log)-expressible. After the components are decomposed we are guaranteed due to the (poly log)-expressibility property that the information needed from each component can all fit onto a single machine (as there are only m machines and they all return polylog sized data) . Thus in a single round on a single machine we can merge all the data starting from the leaf nodes and dynamically compute the value for each node moving up in the tree.

So all that is left to do is to prove our assumption that this problem is (poly-log)-expressible.

(poly-log)-Expressible Proof

Now we we will give an explicit sequential DP in order to prove that the Maximum Weighted Matching satisfies the (poly log)-expressiveness property. Once we show this we can immediately solve using Theorem 1.

A few key definitions: For any vertex $v \in T^b$, $C(v)$ is the maximum weight of an extended matching of the subtree rooted at v where at least one edge from v to one of its children is part of this extended matching. Conversely let $C'(v)$ be the maximum weighted matching value but where *none* of the edges from v to one of its children are present in the matching.

Since we are now working with a binary tree, we can say that for any vertex v it will have two children u_1, u_2. In the easiest case where both children are "real" children as opposed to an auxiliary child, then we can define

$$C(v) = \max\{w_{(v,u_1)} + C'(u_1) + C(u_2), w_{(v,u_2)} + C'(u_2) + C(u_1)\}$$

This is intuitive in terms of the matching problem, as we we are basically computing is it better to choose the edge to u_1 or to u_2 with respect to their two edge weights and in classic Dynamic Programming fashion it depends recursively on the lower levels of the tree. Notice we cannot choose both as these two edges share vertex v.

Conversely we can define

$$C'(v) = \max\{C(u_1), C'(u_1)\} + \max\{C(u_2), C'(u_2)\}$$

$C'(v)$ is essentially calculating if we do not pick either of the edges v, u_1 or v, u_2, what is our maximal possible matching, as it is very possible that a vertex is simply not include in the subgraph M^b as perhaps all of the edges near it are super low value, or it's children both have higher edge values theoretically.

For simplicity we omit the recursive statements in the case where one or both of u_1, u_2 are auxiliary nodes: [BBD+18] covers these cases more rigorously.

Now let us show that $C(v), C'(v)$ imply that the (poly log)-expressiveness property holds for the Dynamic programming problem.

Claim 1. *For any $v \in T$, $\exists$ functions $f_v, f'_v : \{0,1\}^4 \to \mathbb{Z}$ such that:*

$$C(v) = \max_{a_i \in \{0,1\}} \{a_0 C(u_1) + a_1 C'(u_1) + a_2 C(u_2) + a_3 C'(u_2) + f_v(a_0, a_1, a_3, a_4)\}$$

$$C'(v) = \max_{a_i \in \{0,1\}} \{a_0 C(u_1) + a_1 C'(u_1) + a_2 C(u_2) + a_3 C'(u_2) + f_{v'}(a_0, a_1, a_3, a_4)\}$$

This proof is fairly simple as for any combination of a_i, you can set $f(a_i)$ to be whatever it needs to be to make it work. Easy example: if $a = 0110$ this means that we are computing $C(u_2) + C'(u_2) + f_v(a)$, which means for this combination of inputs $f(a) = w_{(v,u_1)}$. Thus since this dynamic problem builds recursively we can treat each $C(v)$ as a variable instead of numbers, doing this allows us to only care about the value of of our top level root node r. So in order to compute maximal weighted matching we simply need to store $f_r, f_{r'}$. Notice, that since these f functions only have $2^4 = 16$ different combinations, so we are required to only store $O(1)$ different potential outputs.

So we have shown that the proposed DP solution to Maximal Weighted Matching satisfies the property that it is (poly log)-expressive, and so we can immediately plug it into Theorem 1 from above and we can say:

Corollary For any given m, there exists an algorithm to solve the Maximum Weight Matching problem on trees in $\tilde{O}(\log n)$ rounds of $\mathcal{MPC}$ using m machines, such that with high probability each machine uses space $\tilde{O}(n/m)$ and runs in linear time with respect to its individual input size.

The size of m can be argued about in a similar way as we did in the closest points section, as we know that at least $s > m$. However, this bound works for any given number of machines so it is very general and globally implemetnable.

3.3 Applications

The above Dynamic Programming approach for Maximum Weighted Matching is very general for graphs and can be applied to a variety of important graph questions. Specifically, for Maximum Independent Set instead of keep track of potential subgraph maximal matching weight, we only need to consider if the root vertex of a subtree is part of the independent set or not, thus the only thing we would have to change for our $C(n)$ is to change it from a max to a sum.

This framework is closely applicable to Maximum Independent Set, Minimum Vertex Cover, Longest Path, and many other DP problems for graphs.

4 Summary and Conclusions

Even though the literature on Dynamic Programming methods in the MRC framework is quite limited, there are several conclusions that can be drawn from the above-surveyed papers. Firstly, we remark that, for many problems, the general structure of Dynamic Programming can be leveraged to obtain $\tilde{O}(\log n)$ $(1+\varepsilon)$-approximate algorithms. As seen in [BBD+18], very general classes of problems with an obvious degree of parallelism (i.e. Dynamic Programming on Trees, for log-expressible or linearly-expressible problems) can, indeed, be efficiently parallelized in the MRC framework without much insight into the structure of the individual problems, but rather using only general properties of the entire class. In contrast, as seen in [IMS17], a thorough understanding of the problem, in addition to some special structure (i.e. monotonicity and decomposability) can yield very efficient $O(1)$-round MRC algorithms, which can become useful in practice, besides any theoretical appeal. The construction for these problems is, at this stage, still very intricate and highly dependent on the individual problem. Lastly, problems which do not satisfy either of these conditions (i.e. monotonicity/decomposability or log/linear-expressibility on trees), such as the Edit-Distance problem, are still difficult to tackle in $O(\log n)$-rounds. In this sense, even though [BEG+18] arrive at an $O(\log n)$-round $(3+\epsilon)$ approximation algorithm with sublinear space and time guarantees, the exponents for time and space are only slightly sub-linear and therefore still prohibitive. This suggests that the conditions of outlined by [IMS17] and [BBD+18] are, in some sense, necessary for the development of efficient algorithms. Of course, an open question is what are the exact requirements that such a problem must satisfy. Even though the authors of these papers provide some more or less formal guidelines, it is still to be expected that these are only particular instances of a more general class of efficiently-solvable MRC problems.

References

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach.* Cambridge University Press, USA, 1st edition, 2009.

[BBD$^+$18] MohammadHossein Bateni, Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, and Vahab Mirrokni. Brief Announcement: MapReduce Algorithms for Massive Trees. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*, volume 107 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 162:1–162:4, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[BEG$^+$18] Mahdi Boroujeni, Soheil Ehsani, Mohammad Ghodsi, Mohammad Taghi Hajiaghayi, and Saeed Seddighin. Approximating edit distance in truly subquadratic time: Quantum and mapreduce. *CoRR*, abs/1804.04178, 2018.

[Coo79] Stephen A. Cook. Deterministic cfl's are accepted simultaneously in polynomial time and log squared space. In *Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing*, STOC '79, page 338–345, New York, NY, USA, 1979. Association for Computing Machinery.

[HLSS19] MohammadTaghi Hajiaghayi, Silvio Lattanzi, Saeed Seddighin, and Cliff Stein. Mapreduce meets fine-grained complexity: Mapreduce algorithms for apsp, matrix multiplication, 3-sum, and beyond. *CoRR*, abs/1905.01748, 2019.

[IMS17] Sungjin Im, Benjamin Moseley, and Xiaorui Sun. Efficient massively parallel methods for dynamic programming. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2017, page 798–811, New York, NY, USA, 2017. Association for Computing Machinery.

[Knu71] D. E. Knuth. Optimum binary search trees. *Acta Inf.*, 1(1):14–25, mar 1971.

[Ukk85] Esko Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64(1):100–118, 1985. International Conference on Foundations of Computation Theory.

[Yao80] F. Frances Yao. Efficient dynamic programming using quadrangle inequalities. In *Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing*, STOC '80, page 429–435, New York, NY, USA, 1980. Association for Computing Machinery.